**Introduction to Transaction Processing**

The concept of transaction provides a mechanism for describing logical units of database processing.

**Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications.

These systems require high availability and fast response time for hundreds of concurrent users.

A transaction is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness.

A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

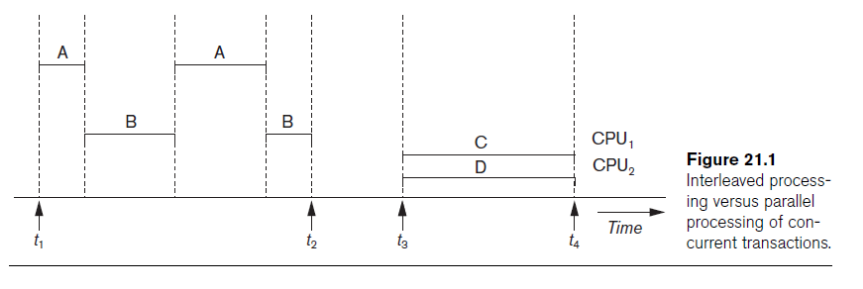
A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.

Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems.

In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system. Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time.

A single central processing unit (CPU) can only execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**, as illustrated in Figure 21.1, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes. If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1.

Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.



**Transactions, Database Items, Read and Write Operations, and DBMS Buffers**

A **transaction** is an executing program that forms a logical unit of database processing.

A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.

A single application program may contain more than one transaction if it contains several transaction boundaries.

If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**.

A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database.

The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name.

Using this simplified database model, the basic database access operations that a transaction can include are as follows:

■ **read\_item**(***X*).** Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X.*

■ **write\_item**(***X*).** Writes the value of program variable *X* into the database item named *X*. the basic unit of data transfer from disk to main memory is one block.

Executing a read\_item(*X*) command includes the following steps:

**1.** Find the address of the disk block that contains item *X*.

**2.** Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

**3.** Copy item *X* from the buffer to the program variable named *X*.

Executing a write\_item(*X*) command includes the following steps:

**1.** Find the address of the disk block that contains item *X*.

**2.** Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

**3.** Copy item *X* from the program variable named *X* into its correct location in the buffer.

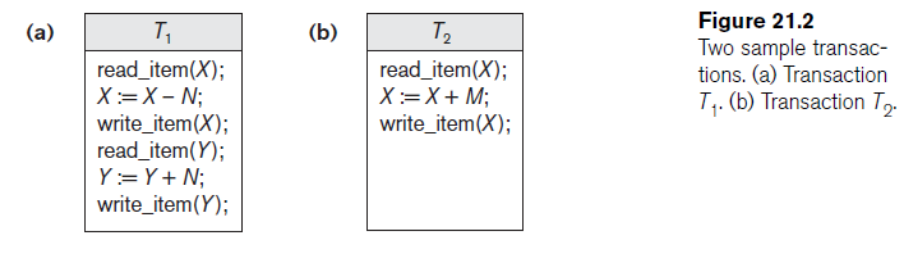
**4.** Store the updated block from the buffer back to disk (either immediately or at some later point in time).

It is step 4 that actually updates the database on disk. In some cases, the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.

The DBMS will maintain in the **database cache** a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed.

When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer replacement policy is used to choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.

A transaction includes read\_item and write\_item operations to access and update the database. Figure 21.2 shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of *T*1 in Figure 21.2 is {*X*, *Y*} and its write-set is also {*X*, *Y*}.



Several problems can occur when concurrent transactions execute in an uncontrolled manner.

We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight.

Each record includes the *number of reserved seats* on that flight as a *named (uniquely identifiable) data item*, among other information. Figure 21.2(a) shows a transaction *T*1 that *transfers N* reservations from one flight whose number of reserved seats is stored in the database item named *X* to another flight whose number of reserved seats is stored in the database item named *Y*.

Figure 21.2(b) shows a simpler transaction *T*2 that just *reserves M* seats on the first flight (*X*) referenced in transaction *T*1.

To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats. When a database access program is written, it has the flight number, flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked.

For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats.

In Figure 21.2(a) and (b), the transactions *T*1 and *T*2 are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items *X* and *Y* in the database.

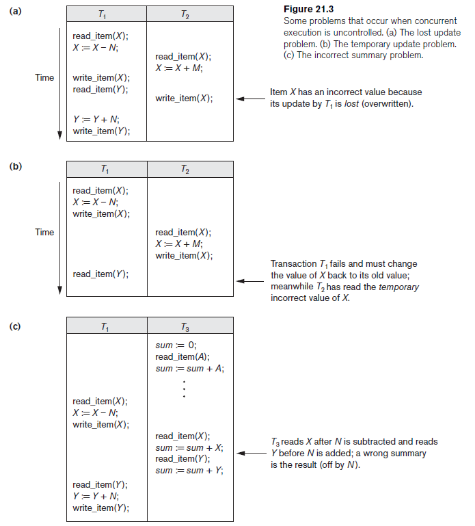
**Why Concurrency Control Is Needed**

**The Lost Update Problem.** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions *T*1 and *T*2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 21.3(a); then the final value of item *X* is incorrect because *T*2 reads the value of *X before T*1 changes it in the database, and hence the updated value resulting from *T*1 is lost. For example, if *X* = 80 at the start (originally there were 80 reservations on the flight),*N* = 5 (*T*1 transfers 5 seat reservations from the flight corresponding to *X* to the flight corresponding to *Y*), and *M* = 4 (*T*2 reserves 4 seats on *X*), the final result should be *X* = 79. However, in the interleaving of operations shown in Figure 21.3(a), it is *X* = 84 because the update in *T*1 that removed the five seats from *X* was *lost*.

**The Temporary Update (or Dirty Read) Problem.** This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 21.3(b) shows an example where *T*1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction *T*2 reads the *temporary* value of *X*, which will not be recorded permanently in the database because of the failure of *T*1. The value of item *X* that is read by *T*2 is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.

**The Incorrect Summary Problem.** If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction *T*3 is calculating the total number of reservations on all the flights; meanwhile, transaction *T*1 is executing. If the interleaving of operations shown in Figure 21.3(c) occurs, the result of *T*3 will be off by an amount *N* because *T*3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it.

**The Unrepeatable Read Problem.** Another problem that may occur is called *unrepeatable read*, where a transaction *T* reads the same item twice and the item is changed by another transaction *T*\_ between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.



**Types of Failures.** Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

**1. A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

**2. A transaction or system error.** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.3 Additionally, the user may interrupt the transaction during its execution.

**3. Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition,4 such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be cancelled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

**4. Concurrency control enforcement.** The concurrency control method may decide to abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

**5. Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

**6. Physical problems and catastrophes.** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

**Transaction States and Additional Operations**

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

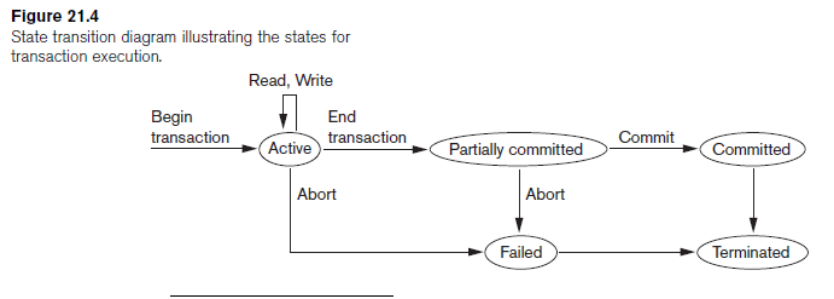
■ BEGIN\_TRANSACTION. This marks the beginning of transaction execution.

■ READ or WRITE. These specify read or write operations on the database items that are executed as part of a transaction.

■ END\_TRANSACTION. This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

■ COMMIT\_TRANSACTION. This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

■ ROLLBACK (or ABORT). This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be **undone***.*



**Desirable Properties of Transactions**

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS.

The following are the ACID properties:

■ **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

■ **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

■ **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

■ **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.

Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems. There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads. And last, the *durability property* is the responsibility of the *recovery subsystem* of the DBMS.